Control Structures in ARM

Implementation of Decisions

- Similar to accumulator instructions
- One instruction sets the flags, followed by another instruction that uses the flags to make the actual branch decision
- ARM compare and test instructions set the flags

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp rn, &lt;op2&gt;</td>
<td>rn - &lt;op2&gt;</td>
<td>Always updates NZCV</td>
</tr>
<tr>
<td>cmn rn, &lt;op2&gt;</td>
<td>rn += &lt;op2&gt;</td>
<td></td>
</tr>
<tr>
<td>tst rn, &lt;op2&gt;</td>
<td>rn &amp; &lt;op2&gt;</td>
<td>Always updates NZ, if shifted. &lt;op2&gt; may affect C flag</td>
</tr>
<tr>
<td>teq rn, &lt;op2&gt;</td>
<td>rn ^ &lt;op2&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Execution

Conditional instructions make a decision to execute based on the state of the appropriate flag. The condition is specified with a two-letter suffix.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Description</th>
<th>Flags tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>Equal</td>
<td>$Z = 1$</td>
</tr>
<tr>
<td>ne</td>
<td>Not equal</td>
<td>$Z = 0$</td>
</tr>
<tr>
<td>cs / hs</td>
<td>Unsigned higher or same</td>
<td>$C = 1$</td>
</tr>
<tr>
<td>cc / lo</td>
<td>Unsigned lower</td>
<td>$C = 0$</td>
</tr>
<tr>
<td>mi</td>
<td>Minus</td>
<td>$N = 1$</td>
</tr>
<tr>
<td>pl</td>
<td>Positive or Zero</td>
<td>$N = 0$</td>
</tr>
<tr>
<td>vs</td>
<td>Overflow</td>
<td>$V = 1$</td>
</tr>
<tr>
<td>vc</td>
<td>No overflow</td>
<td>$V = 0$</td>
</tr>
<tr>
<td>hi</td>
<td>Unsigned higher</td>
<td>$C = 1 &amp; Z = 0$</td>
</tr>
<tr>
<td>ls</td>
<td>Unsigned lower or same</td>
<td>$C = 0 \text{ or } Z = 1$</td>
</tr>
<tr>
<td>ge</td>
<td>Greater than or equal</td>
<td>$N = V$</td>
</tr>
<tr>
<td>lt</td>
<td>Less than</td>
<td>$N \neq V$</td>
</tr>
<tr>
<td>gt</td>
<td>Greater than</td>
<td>$Z = 0 &amp; N = V$</td>
</tr>
<tr>
<td>le</td>
<td>Less than or equal</td>
<td>$Z = 1 \text{ or } N = !V$</td>
</tr>
<tr>
<td>al</td>
<td>Always</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Execution

The two-letter suffix should be used in conjunction with other instructions. For examples:

- `beq`
- `bge`
- `addlt`
- `subgt`

Using a two-letter suffix as a stand-alone instruction is meaningless and will cause a syntax error.
ARM Branch Instructions

Unconditional branch

B (or BAL) branch always

Conditional branches

testing result of compare or other operation (signed arithmetic):

- beq – branch on equal           \((Z==1)\)
- bne – branch on not equal      \((Z==0)\)
- bls  – branch on less than     \(((N \text{ xor } V)==1)\)
- ble  – branch on less than or equal \(((Z \text{ or } (N \text{ xor } V))==1)\)
- bge – branch on greater than or equal \(((N \text{ xor } V)==0)\)
- bgt – branch on greater than   \(((Z \text{ or } (N \text{ xor } V))==0)\)
C if statement in ARM

```c
int x;
int y;
if(x == 0)
    y = 1;
/* assume x is in r0, y is in r1 */
```

<table>
<thead>
<tr>
<th>C Source Code</th>
<th>Inefficient Assembly</th>
<th>Efficient Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>cmp r0, #0</td>
<td>cmp r0, #0</td>
</tr>
<tr>
<td>int y;</td>
<td>beq then</td>
<td>bne endif</td>
</tr>
<tr>
<td>if(x == 0)</td>
<td>then: mov r1, #1</td>
<td>then: mov r1, #1</td>
</tr>
<tr>
<td>y = 1;</td>
<td>// now store r1 in [y]</td>
<td>// now store r1 in [y]</td>
</tr>
<tr>
<td></td>
<td>endif: . . .</td>
<td>endif: . . .</td>
</tr>
</tbody>
</table>
## C if statement in ARM

<table>
<thead>
<tr>
<th>C Source Code</th>
<th>Inefficient Assembly</th>
<th>Efficient Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>cmp r0, #0</td>
<td>cmp r0, #0</td>
</tr>
<tr>
<td>int y;</td>
<td>beq then</td>
<td>bne endif</td>
</tr>
<tr>
<td>if(x == 0)</td>
<td>b endif</td>
<td>then: mov r1, #1</td>
</tr>
<tr>
<td>y = 1;</td>
<td>then: mov r1, #1</td>
<td>// now store r1 in [y]</td>
</tr>
<tr>
<td></td>
<td>endif: ...</td>
<td>endif: ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Note that the "Inefficient Assembly" code contains back-to-back branches (**beq** followed by **b**). We would like to avoid this, since branches may cause a delay slot.
- One way to remove the back-to-back branches is to change the condition for branching, as in the "Efficient Assemble" code.
C if statement in ARM

```c
if(((x+b)>z)
    x+=y;
```

ARM code:

/* assume x is in r0
   y is in r1, and z is in r2 */

```assembly
add r3, r0, r1
cmp r3, r2
ble false // branch to false when((x+y)>z)
           // is false
add r0, r0, r1 // x = x+y
/* now store content of r0 to [x] */
false:
```

(x+y)>z

(true)

(false)

x += y

exit
C if-else statement in ARM

```c
if(i == j)
    f=g+h;
else
    f=g-h;
```

ARM code:

```assembly
/* assume f is in r0, i is in r1, 
  j is in r2, g is in r3, h is in r4, 
*/

cmp   r1, r2          // Z = 1 if i==j
beq   true            // branch to true when i==j
sub   r0, r3, r4      // f = g-h (false)
b     done             // branch to done
true:  add   r0, r3,r4 // f = g+h (true)
done:
```

Diagram:

```
if (i == j)
   f = g + h
else
   f = g - h
```

<table>
<thead>
<tr>
<th>(true)</th>
<th>(false)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i == j</code>?</td>
<td><code>i != j</code></td>
</tr>
<tr>
<td><code>f = g + h</code></td>
<td><code>f = g - h</code></td>
</tr>
<tr>
<td><code>true:  add r0, r3, r4</code></td>
<td><code>done:</code></td>
</tr>
</tbody>
</table>
```
Conditional execution in ARM

An unusual ARM feature is that all instructions may be conditional:

```
CMP r0, #5       // if (r0 != 5)
{
    ADDNE r1, r1, r0    // r1 := r1 + r0 - r2
    SUBNE r1, r1, r2
}
```

- this removes the need for some short branches, improving performance and code density
Implementing Loops

- All for loops, while loops, and do-while loops have an implicit branch from the bottom to the top of the loop.
- This branch instruction becomes explicit when translated into assembly.

```c
while loop
while ( x <= 10 )  /* assume x is in r0
  {              and y is in r1 */
    x = x + y;    
  }
  cmp  r0, #10    // test condition
        bgt  done
  add  r0, r0, r1
  b   loop
done:
```
Loops in C/Assembly

for loop
for ( x = 1; x <= y; x++ )
{
    z *= x;
}

rewritten as while loop
x = 1; 
while ( x <= y )
{
    z *= x;
    x++; 
}

// x: r0, y: r1, z: r2
mov r0, #1
loop:
    cmp r0, r1 // test condition
    bgt done
    mul r2, r2, r0
    add r0, r0, #1
    b loop
done:
Control Structures in ARM

for loop, counted up from 0 to n-1                /* i : r0, n: r1   */
    for ( i = 0; i < n; i++ ) {
        <body>
        mov i, #0  //clear itest:
                loop:
                cmp r0, r1
                bge done
        <body>
        add r0, r0, #1
        b   loop
        done:
    }
rewritten as while loop
i = 0;
while ( i < n )
{
    <body>
    i++;
}
Control Structures in ARM

for loop, counted up from 1 to n
for ( i = 1; i <= n; i++ ) {
   <body>
}

rewritten as while loop
i = 1;
while(i <= n)
{
   if <body>
i++;
}
Control Structures in ARM

for loop, counted down from to n to 1

for ( i = n; i > 0; i-- )
{
    <body>
    mov n, i
}

rewritten as while loop

i = n;
while ( i > 0 )
{
    <body>
    sub i, i, #1
    b loop
}

loop:
cmp i, #0
ble done

done: